APPLICATION FOR PATENT

INVENTORS: VITALY LAGOON and GUY BARRUCH

TITLE: METHOD FOR PROVIDING BITWISE CONSTRAINTS FOR TEST GENERATION

This Application claims priority from US Provisional Application No. 60/228,087, filed on August 28, 2000, which is hereby incorporated by reference as if fully set forth herein.

FIELD OF THE INVENTION

The present invention relates to a method for providing bitwise constraints for test generation, and in particular, to such a method which are operative with a test generation code language such as that provided by the Specman™ functional programming environment (Verisity Design, Inc., Mountain View, California, USA).

BACKGROUND OF THE INVENTION

Design verification is the process of determining whether an integrated circuit, board, or system-level architecture, exactly implements the requirements defined by the specification of the architecture for that device. Design verification for a device under testing (DUT) may be performed on the actual device, or on a simulation model of the device. For the purposes of explanation only and without intending to be limiting in any way, the following discussion centers upon testing which is performed on simulation models of the device.

As designs for different types of devices and device architectures become more complex, the likelihood of design errors increases. However, design verification also becomes more difficult and time consuming, as the simulation models of the design of the device also become more complex to prepare and to test.

The problem of design verification is compounded by the lack of widely generalizable tools which are useful for the verification and testing of a wide variety of devices and device architectures. Typical background art verification methods have often been restricted to a particular device having a specific design, such that the steps of preparing and implementing such verification methods for the simulation model must be performed for each new device.

The process of verifying a design through a simulation model of the device is aided by the availability of hardware description languages such as Verilog and VHDL. These languages are designed to describe hardware at higher levels of abstraction than gates or transistors. The resultant simulated model of the device can receive input stimuli in the form of test vectors, which are a string of binary digits applied to the input of a circuit. The simulated model then produces results, which are checked against the expected results for the particular design of the device. However, these languages are typically not designed for actual verification. Therefore, the verification engineer must write additional programming code in order to interface with the models described by these hardware description languages in order to perform design verification of the device.

Examples of testing environments include static and dynamic testing environments. A static testing environment drives pre-computed test vectors into the simulation model of the DUT and/or examines the results after operation of the simulation model. In addition, if the static testing environment is used to examine the results which are output from the simulation model, then errors in the test are not detected until after the test is finished. As a result, the internal state of the device at the point of error may not be determinable, requiring the simulation to be operated again in order to determine such internal states. This procedure consumes simulation cycles, and can require the expenditure of considerable time, especially during long tests.

A more useful and efficient type of testing is a dynamic testing environment. For this type of environment, a set of programming instructions is written to generate the test vectors in concurrence with the simulation of the model of the DUT and while potentially being controlled by the state feedback of the simulated device. This procedure enables directed random generation to be performed and to be sensitive to effects uncovered during the test itself on the state of the simulation model of the device. Thus, dynamic test generation clearly has many advantages for design verification.

Within the area of testing environments, both static and dynamic testing environments can be implemented only with fixed-vector or pre-generation input. However, a more powerful and more sophisticated implementation uses test generation to produce input stimuli.

One example of such a test generator is disclosed in U.S. Patent Application No. 09/020,792, filed on February 6, 1998, incorporated by reference as if fully set forth herein. This test generation procedure interacts with, and sits as a higher level over, such hardware description languages as Verilog and VHDL. The test generation procedure is written in a hardware-oriented verification specific object-oriented programming language. This language is used to write various tests, which are then used to automatically create a device verification test by a test generator module. A wide variety of design environments can be tested and verified with this language. Thus, the disclosed procedure is generalizable, yet is also simple to program and to debug by the engineer.

The disclosed language features a number of elements such as structs for more richly and efficiently describing the design of the device to be simulated by the model. Unfortunately, the disclosed language and resultant test generation environment does not include the ability to use bitwise constraints.

A bitwise constraint can also be termed a "bit slice constraint". In the modeling of hardware entities such as registers, and input data such as computer instruction streams, there is often a need to express a property over some partial segment of the variable. An example would be a computer instruction word that is 32 bits wide composed of an opcode field of 16 bits and an operand field of 16 bits, where the desired property is to keep the operand field to the value of zero.

U.S. Patent Application No. 09/020,792, described above, discloses a method of applying arithmetic constraints which consider the value of the variable as a whole, or the full 32 bits in the example above. Bitwise constraints add the ability to constrain arbitrary sub-fields of a variable and perform bitwise operations on the variable. Bitwise constraints can be defined as arithmetic relations where at least one of the following bit operations is used:

[:] (slice)

| (bitwise or)

& (bitwise and)

^ (bitwise exclusive or)

~ (bitwise negation)

<< (shift left)

>> (shift right)

One example of such a bitwise constraint is the expression:

keep *x*[2:0] == 0*b*101

Currently available code languages for test generation would compute the operator "[:]" as a function, such that in the above example, an unconstrained *x* variable would first be generated. Next, "*x*[2:0]" would be computed as a function, after which the value for this computed function would be compared to "0*b*101". Such a process would frequently result in an incorrect calculation, and hence an incorrect constraint. A more useful solution would handle such a bitwise constraint correctly through more flexible generation. Unfortunately, such a solution is not currently available.

SUMMARY OF THE INVENTION

The background art does not teach or suggest a method for providing bitwise constraints for test generation code languages, such that these constraints are correctly and flexibly handled.

The method of the present invention overcomes these deficiencies of the background art by enabling bitwise or bit slice constraints to be provided as part of the test generation process, by providing a language structure which enables these constraints to be expressed in a test generation language such as *e* code for example. The language structure for such bitwise constraints is then handled in a more flexible manner, such that the test generation process does not attempt to rigidly "solve" the expression containing the constraint as a function. Therefore, the propagation of constraints in such a structure do not necessarily need to be propagated from left to right, but instead are generated in a multi-directional manner. The language structure is particularly suitable for such operators as "[:]", "|", "&", "^", "~", ">>" and "<<".

According to the present invention, there is provided a method for providing a bitwise constraint for test generation, the method comprising: providing a language structure for expressing the bitwise constraint, said language structure including a plurality of constraint parameters and at least one operator, said constraint parameter constrained to an interval containing at least one value, said interval having interval limits; propagating information bi-directionally to determine interval limits for said constraint parameters; and generating a test value for the constraint parameter.

The method of the present invention could also be described as a plurality of instructions being performed by a data processor, such that the method of the present invention could be implemented as hardware, software, firmware or a combination thereof. For the present invention, a software application could be written in substantially any suitable programming language, which could easily be selected by one of ordinary skill in the art. The programming language chosen should be compatible with the computing platform according to which the software application is executed. Examples of suitable programming languages include, but are not limited to, C, C++ and Java.

BRIEF DESCRIPTION OF THE DRAWING

The invention is herein described, by way of example only, with reference to the accompanying drawing, wherein:

FIG. 1 is a schematic block diagram illustrating an exemplary system according to the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENTS

The method of the present invention enables bitwise or bit slice constraints to be provided as part of the test generation process, by providing a language structure which enables these constraints to be expressed in a test generation language such as *e* code for example. The language structure for such bitwise constraints is then handled in a more flexible manner, such that the test generation process does not attempt to rigidly "solve" the expression containing the constraint as a function. Therefore, the propagation of constraints in such a structure does not necessarily need to proceed from left to right, but instead are preferably generated in a multi-directional manner. The language structure is particularly suitable for such operators as "[:]", "|", "&", "^", "~", ">>" and "<<".

It should be noted that this language structure is the syntactic form that expresses bit constraints, e.g. "keep x[5:8] == 0", and should be distinguished from the computational structure, which supports simple computations with arithmetic and bit constraints and which is a range-list with bit representation according to the present invention. Preferably, the computational structure enables arithmetic and bit constraints to be solved simultaneously, or at least in parallel.

According to the method of the present invention, preferably the computational structure for the bitwise constraints is constructed on the basis of range lists. Each range list includes at least one interval, but may optionally include a plurality of intervals, in which case the range list representing an unconstrained parameter includes one interval containing all possible values . One or more intervals may become invalid during the process of range list intersection. The range lists are then extended with a vector of state values, each of which corresponds to one bit of the generated item. The range list representation is of an arithmetical range, which represents possible values. Each arithmetical value in that range is then preferably mapped to a bit, according to the bit constraint, by translating the number into a value from the vector of state values.

Preferably, each element in the vector has four state values. State values more preferably represent four possible states of the corresponding bit:

0 – the bit is definitely zero;

1 – the bit is definitely one;

$\top$ -- the bit can be either zero or one; and

$\perp$ -- the bit cannot take any legal value, such that a contradiction exists.

For example, if the bit constraint forces the bit to be zero or one, then the corresponding representation is made with the bit set to zero or one. If there is no constraint on the value of the bit, it is set to the "$\top$" operator; if no other option is possible, then the bit is set to "$\perp$", in which no legal value is possible. The computational structure itself preferably includes both interval representation and bitwise representation.

This computational structure can then be used in order to preferably simultaneously represent two different types of constraints. The first type of constraint includes any constraint having an operator as previously defined, which is a bit constraint. The second type of constraint includes a constraint which is related to a constraint of the first type through the propagation of "bitwise" information. The second type of constraint is preferably computed by translating the range list into bitwise values, which take one of the above states of the vector, according to the bit constraint as previously described. Preferably, these different types of constraints are propagated in a bi-directional manner within the bitwise constraint structure; in other words, the effect of resolving arithmetical constraints on bitwise constraints is optionally and preferably assessed, as well as the opposing effect.

The combination of both types of constraints is preferably used for assisting in the resolution of structures related to both types of constraints simultaneously. The combination of these two different structures provides for a more efficient process of resolution, since test generation may optionally feature both bit constraints and arithmetic constraints.

For example, the conversion of bit constraints to range list representation, which would be required if both types of structures could not be resolved simultaneously, could possibly result in an unbounded number of ranges. Similarly, bit constraints alone are not operative to define both types of structures, if range list computational structures were not available, since arithmetical constraints cannot be represented efficiently by bitwise values.

According to preferred embodiments of the present invention, the bitwise constraints are generated by being first reduced, more preferably through the manipulation of the two different types of constraints simultaneously, or at least in parallel. The step of reduction preferably is performed by computing a new value for one of the parameters of the constraint, and then intersecting the range of values of the parameter by this new value, using a bitwise &. For example, given two range lists with corresponding bitwise information, the union of the range lists and the bit information can optionally and preferably be performed, while still maintaining the meaning/effect of the bit constraints and the arithmetical constraints.

More preferably, those bit(s) in the interval which should be zero are indicated by the maximum of the interval.

During the process of value generation with the bitwise constraint structure, preferably the number of solutions for each interval of the interval representation is determined. An interval is then selected randomly according to its weight. Finally, a point is selected from within the selected interval.

According to optional but preferred embodiments of the present invention, an invalid interval may arise when a range list has one or more values that are illegal. Such an interval may optionally and preferably be examined for the effect of other constraints, as well as enabling the constraints on the invalid interval to be handled through the resolution of bitwise constraints in the preferably parallel bitwise structure. In resolving these constraints, the illegal values of the invalid interval may also be resolved by being removed, thereby eliminating the problem of invalid intervals. Such a process may optionally be termed "correction of the interval limit", as the limits of the interval may be changed through this process to remove illegal values, for example. Invalid intervals may also optionally be removed.

The principles and operation of the system and method according to the present invention may be better understood with reference to the drawings and the accompanying description.

Referring now to the drawings, Figure 1 is a schematic block diagram illustrating an exemplary system according to the present invention for test generation. It should be noted that the illustrated system only includes those functions of the test generation procedure which are required for the description of the present invention. A more complete description of the entire test generation procedure may be found in U.S. Patent Application No. 09/020,792, previously incorporated by reference. It should also be noted that although the present invention is described in the context of a simulation model, the present invention is also useful for verification of a physical device. Both the physical device and the simulation model can be described as a DUT (device under test), which is in a test environment.

A test generation system **10** features a simulator **12**, which may accept a design **14** for the device under test (DUT), written in a hardware descriptive language such as Verilog or VHDL. In addition, simulator **12** interacts with a test engine **16** for performing the test generation procedure at run-time. The interaction between simulator **12** and test engine **16** is shown as bi-directional, since test engine **16** provides input to simulator **12**, and in turn receives the results from simulator **12** as input for further operation.

Test engine **16** features a test generator **18**, connected to a run-time system **21** for testing DUT **14** at run-time. Test generator **18** receives one or more constraints **20**, such as the bitwise constraints described in greater detail below, and an I/O data model **22**, and then performs the testing and verification of DUT **14**. Run-time system **21** both drives and samples simulator **12** during the run-time testing procedure. Run-time system **21** also evaluates temporal expressions and emits events. These events are defined according to a temporal coverage event definition input **25**, which feeds the definitions of the events to be covered to run-time system **21**.

During the process of testing and verification, a temporal coverage data collector **24** requests the values for one or more variables from run-time system **21**. These requests are performed according to a triggering event emitted by run-time system **21**, such as a fixed, predefined sampling time and/or the occurrence of a temporal pattern of state transitions as defined by a temporal expression given in a temporal language, for example. Temporal coverage data collector **24** is able to communicate with test generator **18** and to access the requested data through the API (application programming interface) for test generator **18**. Such an API specifies the software function calls required in order to collect the desired data. This collected data is then analyzed by a data analyzer **26**, as described in greater detail below.

After being analyzed by data analyzer **26**, the analyzed data is then displayed to the user, preferably through a GUI (graphical user interface; not shown). For example, the data could be displayed to indicate the presence of particular coverage holes for single coverage items, cross-coverage items and/or interval coverage items. According to a preferred embodiment of the present invention, the analyzed data is used as feedback to adjust constraints **20** according to the coverage afforded by these constraints. For example, if a coverage hole or holes were found, constraints **20** could be adjusted in order to test the absent state or states of DUT **14**. Such feedback is preferably performed automatically, but could also optionally be performed manually by the user.

According to a preferred embodiment of the present invention, constraints **20**, I/O data model **22** and temporal coverage **25** are preferably constructed in *e* code, which is the code language provided by the Specman™ functional programming environment (Verisity Design, Inc., Mountain View, California, USA) and disclosed in U.S. Patent Application No. 09/020,792. Such an embodiment is preferred because of the ease and flexibility of programming in *e* code. The following description centers upon this preferred embodiment, it being understood that this is for the purposes of description only and is not meant to be limiting in any way.

The *e* code language is a hardware-oriented verification specific object-oriented programming language. Objects in this language are instances of "structs", which contain a field, and one or more functions, or methods, which operate on data stored within the field and which interact with other objects. Optionally, a constraint can operate on the field, thereby altering the data stored in the field, for example by restricting the range of possible values for the data. The field can also be used to store more complex structures, including other structs and lists of scalars or structs.

The process of test generation fills data elements, including structs and fields, with random values. The possible data values for each element can optionally be limited by constraints, which provide the direction for the directed test generation. The method of the present invention is particularly directed toward the provision of bitwise constraints. The following section describes a number of illustrative command structures for such constraints in the *e* code language, it being understood that these command structures are only being given for the purposes of illustrating a particularly preferred embodiment of the present invention. In addition, a description is provided of exemplary algorithms for solving such bitwise constraints. This example is illustrative only and is not intended to be limiting in any way. For a more complete explanation of the commands, see U.S. Patent Application No. 09/020,792, previously incorporated by reference.

Examples are provided of the ability of the present invention to handle the following constraints:

$$\textbf{keep} \;\; x[j:i] == y \tag{1}$$

$$\textbf{keep} \;\; x \mid y == z \tag{2}$$

$$\textbf{keep} \;\; x \,\&\, y == z \tag{3}$$

$$\textbf{keep} \;\; x \wedge y == z \tag{4}$$

$$\textbf{keep} \;\; {\sim}x == y \tag{5}$$

$$\textbf{keep} \;\; x << k == y \tag{6}$$

$$\textbf{keep} \;\; x >> k == y \tag{7}$$

Currently each of the operators "[:]", "|", "&", "^", "~", "<<" and ">>" can be only computed as functions. Thus, in all the constraints listed above the information can be propagated only from left to right. As a result some very intuitive user constraints, such as

$$\textbf{keep} \;\; x[2:0] == 0b101;$$

may not be solved by the test generator as known in the background art. The currently known algorithm for such a generator would generate an unconstrained x first, then compute $x[2:0]$ as a function and then check if this value equals $0b101$. Naturally, this strategy would be unsuccessful in $7/8$ of all cases.

The present invention provides a new strategy for more flexible handling of the above constraints. The present invention preferably operates by rendering all of the constraints from (1) to (7) as multi-directional constraints, thereby preferably providing several modes of constraint propagation between constraint parameters.

Initial Assumptions

For the problem (1) it is agreed that i and j are generated before x and y, i.e. there is no constraint propagation from x and y to i and j. Similarly, for the problems (6) and (7) there is no propagation from x and y to k. In practice these restrictions mean that constraints like:

keep $125[j:i] == 0b101;$

keep $14 >> k == 7;$

cannot be solved for i, j or k.

Basic Data Structure: *bit*RL

The data structures for bitwise constraints handling are preferably constructed on the basis of RL's (new range lists). A new structure called *bit*RL is preferably obtained by extending RL type with a vector of 4-state values, each of which corresponds to one bit of the generated item. Values in this vector represent four possible states of the corresponding bit:

0 — the bit is definite zero,

1 — the bit is definite one,

$\top$ — the bit can be either zero or one,

$\perp$ — the bit cannot take any legal value (indicates a contradiction).

The following notation for *bit*RL is used through this document:

$$x = \left\{ \begin{array}{c} [x_1..x_2,\ x_3..x_4,\ \ldots] \\ \langle 0\top110\top00\ldots\rangle \end{array} \right\}$$

The two components of this representation are called respectively *interval representation* and *bitwise representation*.

The following two groups of constraint parameters should be represented by *bit*RLs:

1. parameters of constraints (1) – (7),
2. parameters connected with those from the first group by constraints for which "bitwise" information can be propagated across the constraint.

Algorithms

Reduction

Problems (2), (3) and (5) may optionally and preferably be reduced straightforwardly using Table 1. A basic reduction step consists in computing a new value for one of the constraint parameters and then intersecting the range of this parameter (using bitwise &) by this new value. Full reduction of a bitwise constraint thus consists in successive application of this basic step to each constraint parameter.

$\mid$	⊥	0	1	⊤
⊥	⊥	⊥	⊥	⊥
0	⊥	0	1	⊤
1	⊥	1	1	1
⊤	⊥	⊤	1	⊤

&	⊥	0	1	⊤
⊥	⊥	⊥	⊥	⊥
0	⊥	0	0	0
1	⊥	0	1	⊤
⊤	⊥	0	⊤	⊤

~	
⊥	⊥
0	1
1	0
⊤	⊤

Table 1: Reduction of bitwise operations

Example 1 Consider the constraint **keep** $x \mid y == z$. Consider only the information about bits. Assume that all x, y and z are four-bit ranges with the following initial values:

$$x = \langle 0 \top \top 1 \rangle$$
$$y = \langle 0 \top 0\, 0 \rangle$$
$$z = \langle \top 1\, 1 \top \rangle$$

The reduction is then preferably performed as follows:

- Compute a value of $(x \mid y)$ applying the table for bitwise or separately for each pair of corresponding bits. The result is $z' = \langle 0 \top \top 1 \rangle$
- Compute $(z \,\&\, z')$ which gives $\langle 0111 \rangle$ and assign this value to z. The result is:

$$x = \langle 0 \top \top 1 \rangle$$
$$y = \langle 0 \top 0\, 0 \rangle$$
$$z = \langle 0\, 1\, 1\, 1 \rangle$$

- Using the same table for bitwise or compute the new value $x' = \langle 0 \top 1 \top \rangle$ of x.
- Intersecting x by x' produces:

$$x = \langle 0 \top 1\, 1 \rangle$$
$$y = \langle 0 \top 0\, 0 \rangle$$
$$z = \langle 0\, 1\, 1\, 1 \rangle$$

- Finally the similar steps for y are performed which in this example leaves it unchanged. Thus, the above result is final for this reduction.

Problems (6) and (7) are addressed straightforwardly by shifting the corresponding bitwise representations.

Constraint Propagation Within RLs

The information is preferably propagated in two directions inside *bit*RLs in order to solve both types of constraints in parallel. The maximum of the interval range indicates which bits in the bitwise representation should always be zero.

Example 2 Assume, x is a 6-bit **uint** with the following initial range:

$$x = \left\{ \begin{array}{c} [2..6,\ 10,\ 21..30] \\ \langle \mathrm{TTT1T0} \rangle \end{array} \right\}$$

The maximum of the interval representation is 30, which is 011110. Thus, the most significant bit in the bitwise representation should be zero, i.e.

$$x = \left\{ \begin{array}{c} [2..6,\ 10,\ 21..30] \\ \langle \mathrm{0TT1T0} \rangle \end{array} \right\}$$

Propagation in the opposite direction is preferably used for correction of interval bounds and removing of invalid intervals.

Example 3 If the range used in Example 2 is used, the following facts are observed:

- 21 is an imprecise bound since the range of x contains only even numbers (least significant bit is zero),
- 10 is not included in the range of x because its bitwise representation (001010) does not match the bitwise representation,
- 2 is an imprecise bound since the minimal number matching the bitwise representation is 4 (binary 000100).

For this case the correct result of the propagation from bitwise representation to intervals should be:

$$x = \left\{ \begin{array}{c} [4..6,\ 22..30] \\ \langle 0\top\top 1\top 0\rangle \end{array} \right\}$$

The following example shows the above computations in more detail.

Example 4 Assume the precise lower bound for the interval [21..30] of Example 3 is to be computed. Namely, a minimal number t, such that $21 \leq t$ and t satisfies the bitwise representation $\langle 0\top\top 1\top 0\rangle$.

The computation goes as follows. First, check if the lower bound satisfies the bitwise representation. In this case we are done. However, $21 = (010101)_2$ does not match $\langle 0\top\top 1\top 0\rangle$. Now, consider only those bits of 21 which correspond to $\top$ in the bitwise representation. Namely, the underlined bits from $(0\underline{10}1\underline{0}1)_2$ are used to obtain the number $4 = (100)_2$. The next step is to increment this number getting $5 = (101)_2$ and assign the obtained bits respectively to $\top$-bits in the bitwise representation: the result is $(010110)_2 = 22$.

Similarly, upper bounds are computed by considering all bits corresponding to $\top$ in representation of the upper bound. The only difference is that decrement operation is used instead of increment.

As can be observed, the complexity of evaluation of each bound depends linearly on the number of bits in bitwise representation of range.

In general the process of propagation between bitwise and interval representation consists in computing precise bounds for each interval. Consider an interval $[l..h]$. For the lower bound l a minimal number l' is computed such that $l \leq l'$ and l' matches the bitwise representation. Similarly for the higher bound h, a maximal number h' is computed such that $h' \leq h$ and h' matches the bitwise representation. If the new interval $[l'..h']$ is valid, i.e. $l' \leq h'$, it replaces $[l..h]$. Otherwise the interval $[l..h]$ is removed from the representation.

Generation

In this section the question of value generation out of *bit*RLs is addressed. The following procedure is preferably followed:

- Determine the number of solutions for each interval of the interval representation,
- Select an interval randomly according to its weight,
- Select a point within the selected interval.

The following examples illustrate how to compute the number of solutions for each interval in the interval representation.

Example 5 Continuing with the case of Example 3:

$$x = \left\{ \begin{array}{c} [4..6,\ 22..30] \\ \langle 0\top\top 1\top 0\rangle \end{array} \right\}$$

Consider the interval [22..30]. The binary representations of the interval bounds are respectively $0\underline{10}1\underline{1}0$ and $0\underline{11}1\underline{1}0$. Only the underlined bits are considered and collected, i.e. those which set to $\top$ in the bitwise representation of x. Now, if the underlined bits are collected, two numbers are obtained: $5 = (101)_2$ and $7 = (111)_2$. It's clear that each number satisfying the bitwise representation of x and laying inside [22..30] can be obtained as a bit combination ranging between 101 and 111. There are 3 solutions $(7 - 5 + 1)$ within the interval [22..30].

Using the same technique, the first interval [4..6] is used to compute two valid solutions.

Similarly, as binary representations can be used for computing the number of solutions, they can be used for random selection of points.

Example 6 As it is shown in Example 5, the probability of selecting the interval [2..6] on the second step of the generation algorithm is $2/5$ and the probability of selecting [22..30] is $3/5$ (according to the number solutions in the intervals). Assume that the second step of the algorithm selects [22..30]. Now, the goal is to find a number in [22..30] which satisfies the bitwise representation of x. Each solution for this problem can be expressed as a binary combination of three bits ranging from $5 = (101)_2$ to $7 = (111)_2$. Thus, a number from the interval [5..7] is selected randomly. Assume that the selection is $6 = (110)_2$. The final step consists in taking the three bits of 6 and assigning them to the positions represented by $\top$ in the bitwise representation of x. Thus, the final result is $(0\underline{11}1\underline{0}0)_2 = 28$.

Additional Reordering

In order to increase the number of cases successfully handled by the present invention, the following preferred additional order rules for "bit slicing" and "shift" constraints may optionally and preferably be considered:

$$\textbf{keep } x[j:i] == y \; : \; (i,j) \rightarrow (x,y), \;\; (y) \Rightarrow (x)$$

$$\textbf{keep } x \ll k == y \; : \; (k) \rightarrow (x,y)$$

$$\textbf{keep } x \gg k == y \; : \; (k) \rightarrow (x,y)$$

Here "$\rightarrow$" and "$\Rightarrow$" denote "hard" and "soft" ordering respectively. All "hard" order relations shown above are covered by the initial assumptions . The motivation behind the "soft" ordering $(y) \Rightarrow (x)$ in bit slicing constraint is preferably explained by example.

Example 7 Assume that the following constraint is present:

keep $x[3:0]$ **in** $[1,2,4,8]$;

This constraint naturally means that only one of four least significant bits of x should be 1 and all the others be 0. This constraint is preferably automatically translated into the following two constraints:

keep $x[3:0] == t$;

keep t **in** $[1,2,4,8]$;

where t is a temporary variable. To avoid problems with this example, preferably the implicit "soft" ordering $(t) \Rightarrow (x)$ is used to generate t to one of values from $[1,2,4,8]$. Then the generated value is propagated to the bits of x and the constraint finally succeeds.

Some Examples

The following declarations are assumed through this section:

x : **byte**;

y : **uint(bits** : 2);

Now consider separately the following eight test cases:

1. Given the constraint:

   keep $x[4:3] == 0$;

   This case can be solved by the present invention as shown in the previous sections.

2. Given the constraints:

   keep $x[4:3] == 0;$

   keep $x > 7;$

   Same as in (1).

3. Given the constraints:

   keep soft $x[4:3] == 0;$

   keep $x > 7;$

   Same as in (1). The reduce of "soft" constraint affects the *bit*RL of x exactly as in the case of "hard" constraints.

4. Given the constraints:

   keep $x[4:3] == 0;$

   keep soft $x > 7;$

   Same as in (3).

5. Given the constraints:

   keep $x[5:4] == 0;$

   keep $x[7:6] > 0$ **or** $x[3:0] > 7;$

   keep $x > 7;$

   In this case the generator first reduces the first and the third constraints, which gives the following range for x:

$$x = \left\{ \begin{array}{c} [0x8..0xFF] \\ \langle TT00TTTT \rangle \end{array} \right\}$$

   Any value generated from this range by the present invention is either in [0x8..0xF] or greater than 0xF, in which case it has some of the bits of $x[7:6]$ set to 1. Thus, one of the or-alternatives of the second constraint succeeds.

6. Given the constraints:

   keep $x[4:3] == y$;

   keep gen (y) **before** (x);

   keep soft $y > 0$;

   This test case is solved by the generator as shown in the previous sections. Moreover, the ordering $(y) \rightarrow (x)$ is redundant here because of the implicit ordering $(y) \Rightarrow (x)$ imposed by the first constraint.

7. Given the constraints:

   keep $x[4:3] == y$;

   keep gen (x) **before** (y);

   keep soft $y > 0$;

   This test case may fail in $1/4$ of cases because of user error. The problem here is that the information about the range of y ([1..3]) cannot be captured in the *bit*RL of x. Thus, $x[4:3]$ can be set to 0 if x is generated before y.

8. Given the constraints:

   keep $x[4:y] == 0$;

   By the initial assumption, y is always generated before x. So, y is generated to a value from $[0..3]$ and then the resulting constraint can be successfully solved by the generator.

An example of the actual process for providing and imposing bitwise constraints in the test generation environment is described in greater detail below, it being understood that this example is for the purposes of illustration only and is without any intention of being limiting. The exemplary constrained system is as follows.

First, a struct S, shown below, is assumed to have two fields, X and Y, of 4 bits each. The struct S also is assumed to have three constraints marked (1), (2) and (3) that impose a relationship between X and Y. The constraint (1) is arithmetic, while constraints (2) and (3) are bit-wise constraints.

```
struct S
  field X (4 bits)
  field Y (4 bits)
(1) keep X > 4
(2) keep Y[1:2] == X[1:2]
(3) keep Y[1] == 0
```

The application of constraint (1) to field X results in the following range list, shown in the box below.

```
X = 5..15
```

The application of constraint (3) to field Y results in the following bit range (T represents either 1 or 0), shown in the box below.

```
Y =
<TT1T>..<TT1T>
```

Constraint (2) determines a relationship between field X and field Y. The application of constraint (2) first causes a conversion of the field X range list to bit ranges. The arithmetical values in the range list are preferably converted to bit ranges by first determining bit values for each arithmetical value. If the bit does not change throughout the values of the range list, then the bit is set to either 0 or 1. If the bit does change value throughout the arithmetical values, then the bit is preferably set to the "T" value. The effect of a bitwise constraint may cause the bit to be converted to "0", for example.

$$\begin{tikzcd}
\boxed{X = 5..15} \arrow[r] & \boxed{X = <T1T1>..<1TTT>}
\end{tikzcd}$$

Next, the following reduce step results in *and*ing together the bit ranges of fields Y and X to create a new bit range for field X, as shown in the boxes below.

$$\begin{tikzcd}
\boxed{\begin{array}{l} Y = \\ <TT1T>..<TT1T> \end{array}} \arrow[r, Rightarrow] & \boxed{\begin{array}{l} X = \\ <T111>..<1T1T> \end{array}}
\end{tikzcd}$$

The generation step that follows assigns values in the permissible range, as determined from the previous step, such as setting field Y to 6 and field X to 7, for example, shown in the boxes below.

$$\boxed{Y = 6}$$

$$\boxed{X = 7}$$

While the invention has been described with respect to a limited number of embodiments, it will be appreciated that many variations, modifications and other applications of the invention may be made.